II. Computation and Analysis

SYSTEMS DIVISION

N67-29143

A. Computer Technique for Electron and Ion Gun Design, *N. C. Adams*

The article "Electrostatic Charge Densities on an Electro-optical Star Tracker," by N. C. Adams, E. M. Keberle, and W. Silsby (SPS 37-42, Vol. IV, pp. 10–15) was the first phase of an attempt to evaluate the motion of electrons through an electric field set up inside the star tracker. Several problems arose with this basic approach, forcing further investigation and study. A more computer-oriented method was subsequently discovered which permits the analysis of electron motion through instruments with general-type external boundary conditions. A computer program is currently available for this work.[1]

The problem was approached by replacing the continuous Laplace (or Poisson's) equation by a set of finite difference equations. The region of interest was then overlayed with a fine mesh (Fig. 1), and the difference equation was written for each mesh point. The resulting matrix equation and the equations of motion were then solved.

[1]The program in its original form was obtained from Dr. V. Hamza, Bellcomme.

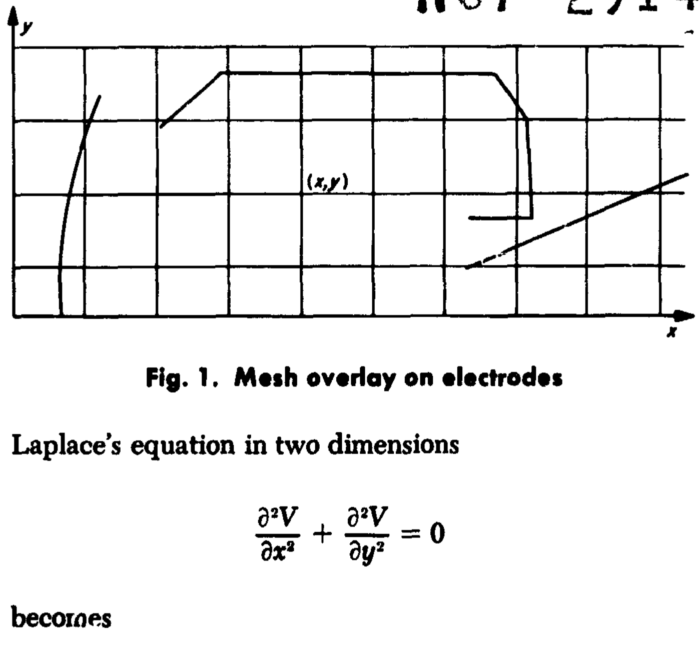

Fig. 1. Mesh overlay on electrodes

Laplace's equation in two dimensions

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0$$

becomes

$$V(x,y) = \frac{1}{4}\,[V(x+h,y) + V(x,y+h) + V(x-h,y) + V(x,y-h)]$$

for any mesh point (x,y).

Poisson's equation in two dimensions

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = -\frac{\rho}{\epsilon_0}$$

becomes

$$V(x, y) = \frac{1}{4}\left[V(x + h, y) + V(x, y + h) + V(x - h, y) + V(x, y - h)\right] + \frac{\rho}{\epsilon_0}$$

By writing the difference equations for each mesh point we obtain N linear algebraic equations with N unknowns. This matrix equation is then solved and numerical differentiation is performed to compute $\partial V/\partial x$ and $\partial V/\partial y$. This yields the force which is required for the equations of motion.

The following is an example of the analysis which can be performed using this program. Given a specific electrode configuration one would like to focus the electrons on an image plate in such a manner as to minimize the dispersion diameter. Different runs can be made varying one design parameter at a time to determine its effect on the focusing. Once a satisfactory dispersion diameter has been obtained in conjunction with an acceptable geometry the design is complete.

At this time the design parameters must be changed manually. Future work will include automating this part of the program. Continued modifications are expected for this program; however, it is available for use in its current state.

B. Perihelion Motion of Mercury, *J. D. Mulholland*

The perihelion motion of the planet Mercury has been a subject of lively interest ever since the discovery, in the mid-nineteenth century, that Newtonian theory failed to satisfy the observational data. Numerous hypotheses were advanced in attempting to account for the discrepancy, but it was not until the development of the general theory of relativity that a reasonably satisfactory explanation was found. This theory predicts a motion of perihelion in excess of the Newtonian value by the amount (Ref. 1)

$$\frac{\delta \tilde{\omega}}{\phi} = \frac{12\pi^2 a^2}{C^2 T^2 (1 - e^2)} \qquad (1)$$

where

$\frac{\partial \tilde{\omega}}{\phi}$ = motion of perihelion per unit of revolution (non-dimensional)

a = semi-major axis

C = speed of light

e = eccentricity

T = orbital period

For the planet Mercury, this amounts to some 43″ per century in excess of the Newtonian prediction. The degree to which this figure satisfies the discrepancy noted above has long been regarded as a strong confirmation of general relativity.

Recently, doubt has been expressed in some quarters that the alleged confirmation is really what it seems. R. H. Dicke (Ref. 2), in advocating an alternative gravitational theory, has discussed the implications for general relativity which attend his measured value of the oblateness of Sun. His computations indicate that the measured solar oblateness requires a motion of 3″ per century in the perihelion of Mercury, which is incompatible with the published comparisons between observations and general relativity theory (Ref. 1, 3, 4).

While there are still some unanswered questions regarding the effect of solar oblateness, it is nonetheless worthwhile to re-examine the contributions to the perihelion motion according to Newtonian theory and that due to Eq. (1). These values are directly related to the adopted values of the planetary masses, for many of which there have been significant redeterminations recently.

I have taken the data given by Clemence (Ref. 1) and revised them linearly with the changes in the adopted values of the masses and their standard deviations. The revised value of the relativity effect is computed using C = 299 792.5 km/sec and elements of Mercury taken from Ref. 5. Table 1 gives the resulting data.

The application of the revised masses appears to have improved the correspondence between theory and observation to an extreme degree. The discrepancy is indicated as less than one-fourth the formal standard error. If we follow Clemence (Ref. 1) and estimate the oblateness effect (based on the rotation of the outer parts of Sun) as being 0″10 ±0″20, then the discrepancy is reduced to

$$0''05 \pm 0''68$$

which is truly overpowering, if one believes it. This seems to indicate that no reconciliation is possible between the existence of a solar oblateness as large as is claimed by Dicke and the soundness of general relativity. I do not, however, regard this as a valid conclusion to be drawn from the present data.

The most obvious of the weak links in the comparison of Table 1 is the value of general precession. The range of values currently in use for this quantity (reduced to 1850.0) is from Newcomb's 5024″53 to Oort's 5025″645. Newcomb's value is probably too low, but the present condition of the theory of motion of Earth is bad enough that we cannot hope for a great deal of confidence in any compromise value, such as that in Table 1.

On the other hand, the planetary contributions are based on planetary theories, and changes in the theories will contribute changes to $\delta\tilde{\omega}$. Such changes have occurred both because of analytic refinement and due to revision of mass values. Linear adjustments in $\delta\tilde{\omega}$ due to mass revisions will not entirel[illegible]nt for the true effect. If Dicke's results are confi[illegible], it will be necessary to recompute the Newtonian components of $\delta\tilde{\omega}$ *ab initio* in order that a reliable comparison of theory and observation can be made.

Table 1. Contributions to perihelion advance (epoch 1850.0)

Newtonian theory		
Planet	**1/mass**	$\delta\tilde{\omega}$
Mercury	6047 000 ± 17000	0″025 ± 0
Venus	408 505 ± 6	277″489 ± 0 ?1
Earth–Moon	328 900 ± 1	90″172 ± 0
Mars	3098 335 ± 600	2″527 ± 0
Jupiter	1047.404 ± 0.02	153″584 ± 0
Saturn	3499.6 ± 0.4	7″301 ± 0
Uranus	22930 ± 6	0″141 ± 0
Neptune	19070 ± 21	0″043 ± 0
		Total 531″282 ± 0.01
General relativity and precession		
General relativity		42″98 ± 0.03
General precession (Duncombe)		5025″32 ± 0.50
Total value, observed value, and discrepancy		
Total value (Newtonian theory + relativity + precession)		5599″59 ± 0.50
Observed value		5599″74 ± 0.41
Discrepancy		0″15 ± 0.65

C. FORTRAN IV Language Extensions: ENCODE/DECODE Statements, *H. L. Smith*

1. Introduction

This article describes an extension to the FORTRAN IV language that includes the ENCODE and DECODE statements implemented for the IBM FORTRAN IV compiler in the 7094 computer. These statements are conceptually associated with the formatted WRITE and READ statements, respectively. In an ENCODE/DECODE operation, no actual input/output takes place; data conversion and transmission take place between an internal buffer and elements specified by a list.

2. Description

In a number of data processing applications, it is necessary to process data at the character level. While the FORTRAN language is word-oriented, some provision is made to process character data in the formatted input/output statements. The ENCODE and DECODE statements utilize the syntax of the FORMAT statement and the subroutines which are normally in memory to process the FORMAT statements. In addition to the compiler changes, modifications to the FORTRAN library subroutines FCNV and FIOH were made to interface with a new subroutine FRRD.

a. ENCODE statement. The general form of the ENCODE statement is

ENCODE (*a*, *b*, *c*, *d*) *list*

where

a is an unsigned integer constant or an integer variable whose value specifies the number of characters per internal record.

b is either the statement number or the array name of the FORMAT statement describing the data being encoded.

c is an array, an array element, or a variable name which specifies the starting location of the internal buffer.

d is an optional integer variable into which will be stored, upon completion of the operation, the number of characters actually generated.

list is as specified for a WRITE statement.

The ENCODE statement causes the data items specified by the *list* to be converted to character strings, according to the FORMAT specified by *b*, and placed in storage beginning at location *c*.

Characters are placed into the buffer, starting with the first character position of the location specified by *c*, in consecutive character positions. When a new record is begun, it starts at the first character position following the previous record; in other words, the second record begins at character $a + 1$. The beginning of logical records, except for the first, can be referenced only if *a* is an integral multiple of the length in bytes of the variable *c*.

If the number of characters generated by the FORMAT statement is greater than the specified size of the record, the extra characters are lost; they are not filled into the following record. If fewer characters are generated than are necessary to fill the record, it is filled out with blanks. The ENCODE operation of filling the unused record with blanks is done after generating the characters. This differs from the formatted (BCD) WRITE operation of only blanking the first three words of the output buffer prior to generating the characters.

If *d* is specified, it will be set to the number of characters generated for all of the records processed, excluding any trailing blanks used to fill out the record.

b. DECODE statement. The general form of the DECODE statement is

DECODE (*a, b, c, d*) *list*

where

a is an unsigned integer constant or an integer variable whose value specifies the number of characters per internal record.

b is either the statement number or the array name of the FORMAT statement describing the data being decoded.

c is an array, an array element, or a variable name which specifies the starting location of the internal buffer.

d is an optional integer variable into which will be stored, upon completion of the operation, the number of characters actually scanned.

list is as specified for a READ statement.

The DECODE statement causes the character string beginning at location *c* to be converted to data items, according to the FORMAT specified by *b*, and stored in the elements specified by the *list*.

Characters are obtained from the buffer starting with the first character position of the location specified by *c*, from consecutive character positions. When a new record is begun, it starts at the first character position following the previous record; in other words, the second record begins at character $a + 1$. The beginning of logical records, except for the first, can be referenced only if *a* is an integral multiple of the length in bytes of the variable *c*.

As with formatted READ operations, if the FORMAT statement requires more characters from a record than are specified by the count *a*, the extra characters are taken from increasing storage addresses following the specified record. The extra characters should not be assumed to be blanks.

If *d* is specified, it will be set to the number of characters scanned for all the records processed, excluding the characters passed over when skipping to the next record.

3. Examples

The following is a method of packing the partial contents of two words into one word. Information is stored in memory as follows:

```
LOCX   SSSSxx   (6 BCD characters/word)
LOCY   xxxxTT
```

To form SSSSTT in storage location LOCZ use the following:

```
  DECODE(6,1,LOCY) TEMP
1 FORMAT(4X,A2)
  ENCODE(6,2,LOCZ) LOCX, TEMP
2 FORMAT(A4,A2)
```

The DECODE statement places the last 2 BCD characters of LOCY into the first 2 characters of TEMP. The ENCODE statement packs the first 4 characters of LOCX and the first 2 characters of TEMP into LOCZ.

The ENCODE statement may be used to compute a field width specification for a FORMAT definition at object time. Assume that in the statement FORMAT(2A6,Ij) the programmer wishes to specify j at some point in the program. The following permits j to vary.

```
      REAL  FMT (2)
        •
        •
      ENCODE (12,1,FMT) J
    1 FORMAT (6H(2A6,I,I1,1H))
        •
        •
      WRITE (6,FMT) A,B,M
```

ENCODE packs the value of J (converted to BCD) with the characters: (2A6,I). This packed FORMAT is stored in locations FMT(1) and FMT(2). The WRITE statement will output A and B under the specification A6 and the quantity M under the specification Ij.

4. Status

Implementation of the ENCODE/DECODE statements is currently operational. The coding is being forwarded to IBM for inclusion in the SHARE Program Library, thus making it available for all 7090/7094 computer installations.

D. Syntactic Processors, *D. A. Germann*

1. Introduction

Computer-program language processors have traditionally developed along the lines of embedding analysis and code generation as an integral part of the processor. Although this generally produces the fastest processor, modifications to the processor are usually difficult and time consuming. Consequently, as a research tool in language development, the traditional processor has very limited use.

To provide the tools needed for the investigation of languages, syntax-directed processors are used, since modifications to the processor are relatively simple. This article will present the fundamentals of syntactic processors, and in particular, discuss two such processors which have proven quite useful.

2. Syntax Notation

The syntax of a programming language is the specification of the structure of the language. In fact, the syntactics of programming languages are quite analogous to that of natural languages. For example, a paragraph could be compared to a procedure or block and a common sentence could be compared to a declaration or assignment statement in a programming language. Just as diagramming natural languages aids in understanding their structure, similar diagramming of programming languages will clarify their structure. One popular form for such diagramming is known as Backus Normal Form (BNF) (Ref. 6). The principal features of a dialect of this form are:

(1) A definition has the form

syntactic-type-being-defined := definition-1 / definition-2 /···

(2) The symbol := separates the element being defined from its definition.

(3) The symbol / separates alternates.

(4) Terminal symbols are enclosed by quote marks.

(5) Non-terminal symbols stand by themselves without quotes.

(6) Definitions may be grouped by the use of parentheses.

(7) Iteration of a definition is indicated by enclosing it in parentheses preceded by a dollar sign.

(8) A null definition is indicated by the symbol .NULL.

As a simple example, consider the following:

person := man / woman

This simply defines a person as being either a man or a woman. Now, assume we wish to define a crowd as being an arbitrary number of persons. Using rule 7 we could write:

crowd := $(person)

As another example, consider the following algebraic expressions:

$$C$$

$$B + C$$

$$A * (B + C)$$

The syntactic equations which define these expressions are as follows:

$$\text{SUM} := \text{TERM } \$(\text{'+' TERM})$$

$$\text{TERM} := \text{PRIME } \$(\text{'*' PRIME})$$

$$\text{PRIME} := \text{VAR} / \text{'('SUM')'}$$

$$\text{VAR} := \text{'A'} / \text{'B'} / \text{'C'}$$

As a final example, the syntax for a subset of FORTRAN IV is shown in Fig. 2.

3. Development of Syntactic Processors

Before discussing syntactic processors, a few definitions need to be introduced (Ref. 7):

(1) *Source language:* The language being described. For example, in Fig. 2 the source language is a subset of FORTRAN IV.

```
          .SYNTAX.
FORTRAN-PROGRAM..=  $( ARRAY-STATEMENT ) $( STATEMENT )   END-CARD
  ARRAY-STATEMENT..=  'DIMENSION'  ARRAY-LST  EOC
    ARRAY-LST..=  SIM-VARIABLE  '(' INTEGER  ')'  $( ',' SIM-VARIABLE '('
                                                      INTEGER ')'   )
  STATEMENT..=  REMARK  /  CODE-STATEMENT
    REMARK..=  'C'
    CODE-STATEMENT..=  LABEL/.NULL  ( FORMAT-STATEMENT / UNLABELED-STATEMENT )
      LABEL..=  INTEGER
      FORMAT-STATEMENT..=  'FORMAT'  '(' FORMAT-SPECS ')'   EOC
      UNLABELED-STATEMENT..=  IF / GOTO / STOP / READ / WRITE / ASSIGNMENT
        IF..=  'IF'  '('  BOOLEAN-EXPRES  ')'  UNLABELED-STATEMENT  EOC
          BOOLEAN-EXPRES..=  SUM  ( REL-OP  SUM )/ .NULL
            REL-OP..=  '.EQ.' / '.NE.' / '.GT.' / '.LT.'
        GOTO..=   'GO' 'TO' LABEL   EOC
        STOP..=   'STOP'
        READ..=   'READ'  UNIT-FORMAT  IO-VAR-LST   EOC
        WRITE..=   'WRITE'  UNIT-FORMAT  IO-VAR-LST   EOC
          UNIT-FORMAT..=   '('  INTEGER  ','  LABEL  ')'
          IO-VAR-LST..=   VARIABLE   $( ',' VARIABLE )
        ASSIGNMENT..=   VARIABLE  '='  SUM   EOC
          SUM..=   TERM  $( AOS TERM )
            TERM..=  PRIME  $( MOD PRIME )
              PRIME..=   AOS/.NULL    ( VARIABLE / CONSTANT / FUNCTION /
                                        '(' SUM ')' )
                AOS..=  '+' / '-'
                MOD..=  '*' / '/'
                VARIABLE..= SIM-VARIABLE  ( '(' SUM ')' )/.NULL
                  SIM-VARIABLE..=  ALPHA  $( ALPHA-NUM )
                CONSTANT..=  INTEGER   ( '.'  INTEGER/.NULL )/.NULL
                FUNCTION..=  NAME  ( '(' SUM   $( ',' SUM ) ')' )/.NULL
                  NAME..=  ALPHA  $( ALPHA-NUM )
                    ALPHA..=  'A' / 'B' / 'C' / 'D' / 'E' / 'F' / 'G' / 'H' /
                              'I' / 'J' / 'K' / 'L' / 'M' / 'N' / 'O' / 'P' /
                              'Q' / 'R' / 'S' / 'T' / 'U' / 'V' / 'W' / 'X' /
                              'Y' / 'Z'
                    DIGIT..=  '1' / '2' / '3' / '4' / '5' / '6' / '7' / '8' /
                              '9' / '0'
                    INTEGER..=  $( DIGIT )
                    ALPHA-NUM..=   ALPHA / DIGIT
  END-CARD..=  'END'  EOC

          .END
```

Fig. 2. BNF description of a subset of FORTRAN IV

(2) *Metalangvage:* The describing language; that is, the language used to describe the source language. In Fig. 2 the metalanguage might be considered that dialect of BNF described in Subsection 2.

(3) *Target language:* The language produced from the source language by the language processor. Usually, this is an assembly language or actual machine code.

Now, having been introduced to the terminology, assume we wish to create a language processor to translate source language A to target language B for machine X. The classical steps to accomplish this would be:

(1) Completely specify source language A, perhaps using the BNF notation or another convenient metalanguage.

(2) Determine the relation between source language A and target language B.

(3) Hand-write the processor to accomplish this translation using an existing language on machine X.

As may be expected, this is not a trivial task; at best, it is both time consuming and quite prone to errors. A more desirable method would be to have a "universal" processor which will accept a description of the source language and proceed to parse and generate code according to this description. This is exactly the philosophy of syntax-directed processors. To further clarify the differences between a conventional language processor and a syntax-directed processor, the processors are diagrammed in Fig. 3 (Ref. 8).

At this time this mystical "universal" processor, along with its metalanguage input, will be developed. This processor may be considered as a simulator of a special machine. This meta machine (Ref. 9), described in Tables 2 and 3, contains only those instructions useful in translating a source language into its target language. In fact, using these instructions, converting from the BNF notation to an assembly code is almost immediate. For

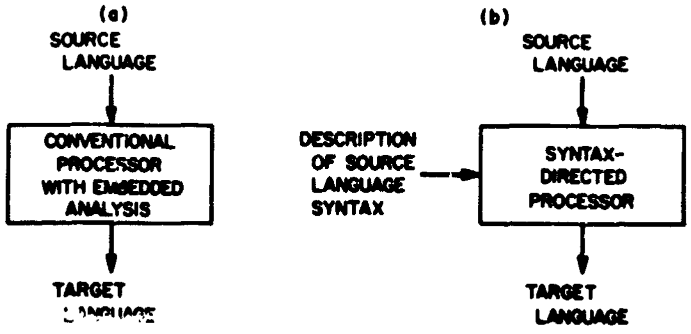
(a)
SOURCE LANGUAGE
CONVENTIONAL PROCESSOR WITH EMBEDDED ANALYSIS
TARGET LANGUAGE
(b)
SOURCE LANGUAGE
DESCRIPTION OF SOURCE LANGUAGE SYNTAX
SYNTAX-DIRECTED PROCESSOR
TARGET LANGUAGE

Fig. 3. Processors: (a) conventional; (b) syntax-directed

Table 2. Description of meta machine

Item	Description
INBUF	An input buffer containing a continuous string of input characters
OUTBUF	An output buffer to be punched and printed
STACK	A push-down stack
FLAGn	Logical flags (o-n) used for program control

Table 3. Subset of meta machine instructions

Operation	Argument	Instruction
TST	STRING, FLAGn	Compare the first non-blank string in the input stream to the string given as an argument. If the strings are identical, delete the string from the input and set FLAGn true. Otherwise, set FLAGn false.
ID	FLAGn	Test if the next non-blank input string is an identifier. If so, delete it from the input, placing it as the top element of STACK, and set FLAGn true. If not an identifier, set FLAGn false.
CALL	AAA	Enter the procedure at location AAA.
RET	AAA	Return to the procedure which last called AAA.
SET	FLAGn	Set FLAGn true.
RESET	FLAGn	Set FLAGn false.
B	AA	Branch unconditionally to location AA.
BT	AA, FLAGn	Branch to location AA only if FLAGn is true.
BF	AA, FLAGn	Branch to location AA only if FLAGn is false.
PRINT	STRING	Print out the string.
TEMP	n	Create a symbol of form 'TMPn' and place it in STACK.
INC	N, X	Increment N by X.
POP		Pop the top element in STACK.
LAB		Reset the output pointer to column 1.
OUT	(STRING/*n)	Output either the literal string or the nth element in the STACK. Output pointer is reset to column 8.
STOP		Program stop.
END		Denotes end of program.

example, the four syntactic equations presented in Subsection 1 describing a class of algebraic equations are coded below.

```
SUM      CALL   TERM
         TST    '+',FLAG1
         BT     SUM,FLAG1
         RET    SUM

TERM     CALL   PRIME
         TST    '*',FLAG1
         BT     TERM,FLAG1
         RET    TERM

PRIME    ID     FLAG1
         BT     PRIM1,FLAG1
         TST    '(',FLAG1
         BF     BADERR,FLAG1
         CALL   SUM
         TST    ')',FLAG1
         BT     PRIM1,FLAG1
         PRINT  'MISSING RIGHT PAREN'

PRIM1    RET    PRIME

BADERR   PRINT  'UNRECOVERABLE ERROR'
         STOP
```

Note that although the source language is completely described, no consideration for the target language (step 2, above) has been made. The exact details of this translation are much too involved to be included in this article; hence only the results are shown below.

```
SUM      CALL   TERM
         TST    '+',FLAG1
         BF     SUM2,FLAG1
         BF     SUM1,FLAG3
         CALL   SAVE

SUM1     CALL   SUM
         CALL   OUTADD

SUM2     RET    SUM

TERM     CALL   PRIME
         TST    '*',FLAG1
         BF     TERM2,FLAG1
         SET    FLAG3
         CALL   TERM
         CALL   OUTMPY

TERM2    RET    TERM

PRIME    ID     FLAG1
         BT     PRIME2,FLAG1
         TST    '(',FLAG1
         BF     BADERR,FLAG1
         CALL   SAVE
         CALL   SUM
         TST    ')',FLAG1
         BT     PRIME2,FLAG1
         PRINT  'MISSING RIGHT PAREN'

PRIME2   RET    PRIME

BADERR   PRINT  'UNRECOVERABLE ERROR'
         STOP

SAVE     BF     SAVE1,FLAG2
         INC    N,1
         TEMP   N
         OUT    ('STO ',*1)
         RESET  FLAG2
         RESET  FLAG3

SAVE1    RET    SAVE

OUTADD   BF     OUTAD2,FLAG2

OUTAD1   OUT    ('FAD ',*1)
         POP
         RET    OUTADD

OUTAD2   OUT    ('CLA ',*1)
         POP
         SET    FLAG2
         B      OUTAD1

OUTMPY   BF     OUTMP2,FLAG 2
         OUT    ('XCA')

OUTMP1   OUT    ('FMP ',*1)
         POP
         RET    OUTMPY

OUTMP2   OUT    ('LDQ ',*1)
         POP
         B      OUTMP1
         END
```

At this point, let us review briefly what we have accomplished. Basically, all we did was to create a simulator for a machine designed for language translation. Next we encoded the source language description (and its relation to the target language) into a form acceptable to the simulator. The next logical step is to allow the input to be in a form familiar to the programmer, namely in Backus Normal Form. This is easily accomplished since, due to its simplicity, the recognization syntax may be given in only five statements.

```
BNF-PROGRAM := '.SYNTAX.' $(STAT) '.END'
  STAT := IDENT '  ' EXP1
    EXP1 := EXP2 $('/' EXP2)
      EXP2 := EXP3 $(EXP3)
        EX3 := IDENT / STRING /
                 '(' EXP1 ')' / '$' ')' EXP1 ')'
```

4. Practical Considerations of Syntactic Processors

As stated in the introduction, two syntactic processors have been used extensively. These two, TMG and META, differ primarily in the following aspects: (1) backup capability, (2) output flexibility, and (3) dictionary and classification ability.

By backup capability, I mean the ability to rescan an input stream. For example, consider the following assignment statement:

IF(J) = I + K

Using the syntax of Fig. 2 and the meta machine discussed in Subsection 3, the statement would have beer erroneously recognized as an "IF" statement. Thus, when the equal sign is found, an error message would be issued. To avoid this difficulty, we could resort to a keyword language requiring that certain key words not be used as variables. A better solution would be to use backup; that is, after finding that the statement is not correct, the input pointer is backed up and the alternate is called.

The differences in output flexibility are more subtle. In META the output occurs at the same time as the scanning or parsing, whereas in TMG the output is accumulated in an output tree and all output occurs when the parsing is complete. The primary value in postponing the output is the ability to pass arguments between the various output definitions and the ability to perform a post analysis before the actual output.

The third difference, the dictionary and classification ability, can be best illustrated by the following example:

J = WHAT(I)

Using the meta system, it is unknown whether "WHAT" is an array variable or a function reference. Clearly it is desirable to retain certain attributes about the variables. For this reason, TMG has provided for retaining up to 18 bits of information for each variable found.

References

1. Clemence, G. M., *Reviews of Modern Physics*, Vol. 19, pp. 361–364, 1947.
2. Dicke, R. H., *Physics Today*, Vol. 20, pp. 55–70, January 1967.
3. Duncombe, R. L., *Astronomical Papers of the American Ephemeris*, Vol. 16, Part 1, 1958.
4. Wayman, P. A., *Quarterly Journal of the Royal Astronomical Society*, Vol. 7, pp. 138–156, 1966.
5. Allen, C. W., *Astrophysical Qualities*, 2nd Ed., University of London, 1963.
6. Backus, J., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," *Proc. First Intern. Conf., Information Processing*, UNESCO, Paris, France, 1960.
7. Rubinoff, M., *Advances in Computers*, Academic Press, New York, 1966.
8. Ingerman, P. Z., A *Syntax-Oriented Translator*, New York, Academic Press, 1966.

References (contd)

9. Schorre, D. V., "Meta II, A Syntax-Oriented Computer Writing Language." *1964 Proceedings, ACM*, pp. D1.3-1–D1.3-11.

10. McClure, R. M., "TMG-A Syntax-Directed Compiler," *1965 Proceedings, ACM*, pp. 262–274.